Kn support for Knative eventing

Dr. Roland Huß, Matthias Wessendorf

Version v0.1.0, 2019-07-04

Table of Contents

tl;dr

This document is a proposal about how Knative eventing support can enter `kn`, the Knative client. This support should be on the same level and providing the same user experience as the existing support for Knative serving.

It is important to note that everything described here is based on the existing Knative eventing API 0.7 and could be implemented immediately without requiring any changes on Knative eventing.

In this document we are trying to map the scenarios described in this GitHub issue and scenario document to the uses case defined later in Use Cases.

The three key themes of this document are

	Introducing the ***Integrator*** and ***Developer*** role and how their use cases map to `kn` commands	Roles
	How to **extend** importers and channels with custom types that are initially unknown to `kn`	Custom types
	How to help *Developers* with an **opinionated way** to connect their services to a given event topology.	Service connections

Roles

For Knative eventing, there are three roles for which use cases can be classified:

- **Administrators** install Knative on a target platform like Kubernetes. *Administrators* are responsible for registering the Knative Eventing core CRDs and installing operators for watching those. *Administrators* are also responsible for installing any importer (aka source) and channel **types** (CRD and controllers) so that *Integrators* can create integration and channel **instances** with a specific configuration. There can be more than one instance for each type (e.g. different Cron importers firing on different schedules)
- **Integrators** are responsible for managing importer and channel instances out of the list of available types. They are setting up the event topology also responsible for creating brokers, which reference specific channels.
- **Developers** are using the importers, channels and brokers by linking them together, potentially also with Knative services created by the developer.

Of course, a single person can fulfil multiple roles.

`kn` is supposed to provide support for use case for the roles of an *Integrator* and *Developer*. *Administrators* _are out of scope for the Knative client as `kn` is agnostic of the way how Knative is installed.

- For *Integrators* the full eventing feature set should be available so that the event topology can be setup flexibly. The commands for the *Integrator* use cases is a thin wrapper over the underlying custom resources (Channels, Importers and Brokers). The benefit of this approach is not only a simplified resource management but also how `kn` can deal with custom implementations of importers and channels. For this, a particular plugin architecture is proposed in Custom types.
- For *Developers* a more opinionated approach is taken, shielding the developer from direct custom resource management. There are currently three different ways how a service can be registered for events: Direct linkage to an importer, subscription to a channel, or subscription to a broker with a trigger. All of these different ways are to be shielded behind a common UX. In Service connections a suggestion is made how this can be achieved with a "connection" command. For Sequences a workflow is suggested to build up such a sequence of connected transforming services.

The roles defined in the scenarios of the scenario document can be mapped to the roles like:

FaaS Scenario

The "developer" role in the FaaS scenario is also the *Developer* role in this document. It's a role which merely wants to use a given infrastructure and event topology by connecting her services to event producers. See Service connections how the main developer use case is covered.

Event-Driven Scenario

The "event-producer developer" role maps to the *Integrator* as it's the event-producer who is responsible for managing importers and possibly brokers. See Importers, Channels and Brokers for this roles' use cases. The "even-consumer developer" is a *Developer* and is similar to the

"developer" in the FaaS scenario as it's about connecting a service with an event producer (trigger, broker or directly via a channel subscription). Service connections (and also Sequences) hold the use cases for this scenario.

Black-box integration scenario

The "central platform team" is represented by the *Integrator* role for the provisioning part of the black-box software when it is about creating the importer resources. A "central platform team" very likely also has to perform tasks as an *Administrator* to install the software's backend parts. Then, the "developer" in this scenario can use the importers installed by the *Integrator* by managing Service connections). For this black-box scenario adding Custom types is crucial.

Level of Abstraction

Every feature of Knative eventing and serving can be managed with a general purpose CLI tool like `kubectl`. However, it is the goal of `kn` as specialized CLI tool to make frequent use cases easy to perform. Therefore `kn` takes a somewhat opinionated approach, and restricts individual variations and combinations. E.g. there is no support for creating `Routes` directly with `kn` in favour of doing everything over the `Service`.

The question is how much abstracted (or "opinionated") the core Knative concepts should be. This is not a discrete choice but a spectrum with exposing Knative CRs directly on the one side and using a completely new vocabulary specific to `kn` on the other side.

Let's have a look on both ends of the spectrum:

- **Direct resource management** only provides a thin layer of creating a CR directly. *Integrator* use cases tend toward this end for managing the event topology with importers, channels or brokers.
- **Use-case backed interface** encapsulates the management of Knative custom resources behind an opinionated user interface. Examples for this higher-order abstractions are:
  - `kn rollout --strategy blue-green` for creating
  - `kn rollout --strategy canary --canary-target=2 --image=myservice:2.0` for creating a new revision with for a new version with a target of %2 per cent for the canary.
  - `kn connect --source github://secretname@github.com/user/helloworld --select … --destination service://myservice` for connecting an importer to a broker, creating a trigger with `--select` which references a Knative service `myservice` as a sink. ++ *Developer* use cases are opinionated and make the underlying custom resources more or less opaque.

An interesting exception of this mapping here is the `kn service` command, which implements a *Developer* user story. However, since a Knative `Service` is already a developer-friendly abstraction and an umbrella resource for managing other resources (configuration, revisions, routes), it already has the proper abstraction to be used directly by `kn`. There is no similar umbrella object for Knative eventing. Although this is a *Developer* task having this slight abstraction over managing a Service directly via `kubectl` has benefits, because it provides much value as it encapsulates domain knowledge how to create the underlying custom resource. A good example here is the options `--concurrency-limit` and `--concurrency-target` to `kn service create` which are kind of hard and soft limits for when to trigger an autoscaling event, but on the CR they end up in different places (direct custom resource field vs metadata annotation)

Use Cases

All the use cases in this section are crafted with this `kn` UI scheme in mind:

kn <noun> create <name>

Create a <*noun*> identified by <*name*>

kn <noun> update <name>

Update a <*noun*> identified by <*name*>

kn <noun> show <name>

Show details of the <*noun*> instance with name <*name*> [1]

kn <noun> delete <name>

Delete an instance of <*noun*> with <*name*>

kn <noun> list <prefix>

List entities. If <*name-prefix*> is given, filter the entity names on this prefix.

<*noun*> can be either directly reflecting the underlying Knative custom resource (typical for *Integrator* based use cases) or more abstract, developer-oriented, concepts like the proposed `connection` which describes any connection from a `Service` to the event backend. See Service connections for details.

Also, when there is a (hierarchical) relationship between <*nouns*> (like between `service` and `revision`) particular option might filter on the high-level <*noun*> (like in `kn revision list --service myservice`).

This scheme, which has been applied successfully for managing Knative serving, should be preserved for Knatice eventing support as well.

It is to be discussed whether the scheme should be relaxed for supporting developer workflows more naturally, e.g. like in

```
kn rollout
kn rollback
kn connect <service> --broker mybroker
kn disconnect <service> --all
kn split revision1:10% revision2:90%
```

so, in the general form `kn <verb>` where verb concretely refers a developer use case which is not mapped 1:1 to entities (so more of category *Use-case backed interface*)

Moreover, a mixed format could be possible as well. E.g. creating and removing connections with `kn connect` and `kn disconnect`, but listing, updating and showing connections with `kn connection list`, `kn connection update` and `kn connection show`.

Integrator use cases

The following use cases can be categorized by this epic use case below. So they are all about setting up the event topology which includes brokers, channels and the importers that then can be used by a *Developer*.

As an *Integrator* I want to manage importers (sources) and the infrastructure elements like brokers and channels to set up the eventing topology.

The following use cases are a breakdown, how the event topology can be managed by directly managing the underlying Knative eventing resources.

Channels

Channels are used for connecting importers/source to services and provide the backbone for the eventing system. They can be created implicitly via brokers, but they can also be created directly by *Integrators* so a *Developer* can subscribe a service to it.

A channel has a specific type which determines how events are persisted and distributed. There is a set of predefined types but not all are available out of the box on every installation of Knative eventing. The only channel type that is always available is an `in-memory` type. Other types, like `kafka` for a Kafka backed event transport, need extra installation efforts by an *Administrator*. Also *Administrators* can introduce new custom channels which are not known in advance by `kn`. To use these custom channels, a plugin architecture is proposed in Custom types.

One critical use case for the *Integrator* is to list all available types (installed well-known and custom types) that can be used for creating a channel. Let's have a look at this use case first.

As an *Integrator* I want to find all channel types which are available by a given Knative installation

```
# List all channel types which are installed on the cluster and for
# which client support is available
$ kn channel types

TYPE                DESCRIPTION
in-memory           Non-persistent in memory channel (default)
kafka               Kafka backed channel
pubsub              Google Cloud pub-sub
activemq            ActiveMQ backed channel
```

Only those types which can be used for the given Knative installation must show up here. For the four directly supported channel types *in-memory*, *kafka*, *pubsub* and *natss* the corresponding cluster features needs to be enabled by the *Administrator*. If a well-know type is not installed on the cluster, that type won't show up in the list of available types. In our example this is the case for the `natss` type, because no support has installed in the cluster for it. For custom channel types like, e.g. the *activemq* type in this example, also a local **channel plugin** needs to be present. See Custom types for more details on how channel type detection and channel plugins are supposed to work.

As an *Integrator* I want to create a channel with a specified type

```
$ kn channel create mychannel --type kafka --num-partitions=4 --replication-factor=3
```

The `channel create` command creates a channel directly with the given type. If no type is given then the default type is used (typically `in-memory`, but depends on the cluster configuration).

In addition each type has specific configuration options (`--num-partitions` and `--replication-factor` in this example). The client verifies which options are available depending on whether its a well-known type or a custom type:

- For well-known types known to a vanilla Knative eventing installation, the possible options are included in kn.
- For custom types, which are backed by a custom channel plugin, the plugin is called to get the possible options. This process is described in Custom types.

For the user, this difference doesn't matter, so on the UI surface, well-known and custom types are treated the same.

As an *Integrator* and as a *Developer* I want to list all channels

```
# List all channels for the current namespace
$ kn channel list

NAME             TYPE       BROKER  SUBSCRIBERS STATUS     INFLIGHT EVENTS
channel-1        kafka              2           Up         0        34326
myotherchannel   in-memory  default 4           Up
```

This will list all channels available along with some summary description like the channel type, whether it's created on behalf of a broker, the status, the number of subscriptions attached to this channel

If possible some statistic informations would be nice to have, too. E.g. one could show how many events have passed the channel or how many events have not been delivered yet.

As an *Integrator* and as a *Developer* I want to see the details of a channel

```
# Show specific details for a channel
$ kn channel show channel-1

Type: kafka
Broker: default
Subscribers:
- service1 [direct]

Triggers:
- myotherservice [event.type="bla"]
```

Any detail information available, also from related objects, should be shown here. This command is also useful for *Developers* as it helps in understanding the event topology.

As an *Integrator* I want to remove a channel

```
# Remove a channel but check whether it's in use
$ kn channel remove channel-1
```

This command will remove a named channel, but only those who are not managed by a broker. Also, it should be checked whether the channel has some active subscriptions. If this is the case, then by default, an error must be returned. However, an *Integrator* can use `--force` to remove the channel **and** any active subscriptions.

Importers

Importers are the new name of the resources formerly known as "Sources". Please see this document for the motivation for this naming change.

Importers are there to pump events into the eventing topology. Each importer has a specific type, much like channels. In fact, from an implementation's point of view, importers can be treated the same as channels. Moreover, also from a UX point of view, the user interface for both can be nearly the same. However, let's have a look.

As an *Integrator* I want to find out all importer types available so that I know what importers I can create

```
# List all well-know as well as custom importers
$ kn importer types

TYPE            DESCRIPTION
kafka           Kafka importer picking up event from a Topic
kubernetes-api  Import Kubernetes event
cron            Periodic event from a cron importer
twitter         Import tweets by user or search
```

As for Channels there can be well-known importers (e.g. kafka) but also custom importers (twitter).

For full details for how to handle custom types and seamlessly integrate with the well-known types can be found in Custom types.

As an *Integrator* I want to create a new importer so that a *Developer* can use it

```
# Create an importer which picks up Tweets mentioning "knative"
$ kn importer create twitter-knative --type twitter --search knative
```

The mandatory flag for an importer is `--type` which specifies the type to use. The value given must

be one out of the list as given by `kn importer types`.

All other options are specific to the importer's type, much like the type of a channel.

An addition could be to provide here already a `--service` to create the connection to a service, but for the sake of conciseness creation of this connection should be left to `kn connection create` (or `kn connect` if we opt for a verb based flow for *Developer* use cases as described in detail in Use Cases,

As an *Integrator* or *Developer* I want to list all existing importers

```
# List all created importers
$ kn importer list

NAME                TYPE         RESOURCE
twitter-knative     twitter      twittersource.importers.k8spatterns.io
all-seconds         cron         cronjobsources.sources.eventing.knative.dev
```

As an *Integrator* or *Developer* I want to see the details of an importer

```
# Show details for a specific importer
$ kn importer show twitter-knative

Name:            twitter-knative
Resource:        twittesource.importers.k8spatterns.io
Type:            twitter
Search:          knative
Last Checked:    2019-07-04 04:50:12

Broker:          default
Subscribers:
- ....

....
```

As expected `kn importer show` will show all the details for an importer. This is a human-readable output, and specific to the importer's type. For custom types, this output comes directly from kn importer plugin (`kn-importer-twitter` in this case).

As an *Integrator* I want to delete an importer

```
# Delete an importer
$ kn delete importer twitter-knative
```

Deletion should check, whether this importer is still in use. If so, an error should be returned. An *Integrator* can still delete an importer with the option `--force`. In this case, all subscriptions should be removed as well.

Brokers

As an *Integrator* I want to create a broker in a namespace so that a *Developer* can use it

```
# Create a broker
$ kn broker create --provisioner gcp-pupsub
```

Creating a broker will create a resource of kind `Broker` with possible configurations fields offered as an option, like `--provisioner` to specify the cluster channel provisioner for the channel template included by the broker.

As an *Integrator* or *Developer* I want to list all brokers in a namespace

```
# Return an overview of all brokers installed
$ kn broker list

NAME      STATUS      SUBSCRIPTIONS
default   Up          4
mybroker  Up          2
```

As all list commands, it should be possible to export the list of brokers in a machine-readable format like `json` or `yaml`, and it should be possible to filter on brokers to show (startWith filtering).

As an *Integrator* or *Developer* I want to see the details of a broker

```
# Show the details of broker `mybroker`
$ kn broker show mybroker
Name: mybroker
Status: Up

Subscriptions:
- name: my-service-trigger
  type: dev.knative.foo.bar
  service: myservice
- name: other-trigger
  type: prod.knative.foo.bar
  service: prodservice

Importers:
- name: financial-kafka-source
  type: kafka
```

This command should reveal all details of the `Broker` resource itself, but also information about objects that are *referencing* this broker, like the importers which feed events into this broker.

As an *Integrator* I want to delete a broker

```
# Delete broker 'mybroker'
$ kn broker delete mybroker
```

Before deleting a broker, `kn` should check if the broker is still in use. E.g. when there are subscriptions to this broker via triggers, then kn should refuse to delete the broker. However, when an option `--force` is given, then the broker **and** all triggers referencing this broker should be deleted.

Developer use cases

The developer is the user of the eventing topology. She creates services (presumably Knative serving services) and connects them to importers either directly, via a channel or a broker.

As a *Developer* I want to use the eventing topology to receive events for which I can register my services with filtering and chaining.

Service connections

There are several ways how a service can be registered for retrieving cloud events: direct, via a broker or subscription. Depending on the mode, custom resources created looks quite different as well as the preconditions. However, this should not matter for the UI as they all serve the same use case, but with different capabilities.

As a *Developer* I want to connect a service to the eventing infrastructure

```
# Connect a service directly to an importer, giving it a name
$ kn connection create myconnection --service myservice --importer k8sapievents

# Alternative syntax:
$ kn connection create myconnection --service myservice --target importer:k8sapievents

# Alternative syntax (starting from "service")
$ kn service connect myservice --connection myconnection --target
importer:k8sapievents

# Connect a service to a broker with a trigger and the given filter
$ kn service connect myservice --broker default --filter <filter-expression>

# Alternative verb based syntax (see discussion in "Use Cases"):
$ kn connect --service myservice --target importer:ks8apievents

Connection myservice-001 has been created.
```

As seen above, depending on which style to chose, the are three possibilities for the create command:

- Noun-based on "connection": `kn connection create <connection-name> --service <service-name> --target ⋯`
- Piggy-back on "service": `kn service connect <service-name> --connection-name <connection-name> ---target ⋯`
- Verb based: `kn connect --service <service-name> --connection-name <connection-name> --target`

It is worth noting, that the *piggy-back on service* variation above only work smoothly for create a connection (and maybe removing with "disconnect"). All other CRUD operations (show, list, update) should go to the stand `kn connection <verb>` mode, as otherwise this would lead to ugly command names like `kn service show-connection` or `kn service list-connections`).

A possible benefit of *piggy-back on service* and *verb based* would be that the connection name could be auto generated. For the *noun-based on connection* mode this is not possible as it would break the general scheme. However, whether auto generation of names is desirable, is another question.

Regardless what syntax to chose, depending on the `--target` argument, the service is connected to the event system in different ways:

- Directly to an Importer (`--importer <importer-name>` or `--target importer:<importer-name>`)
- With a subscription to a channel (`--channel <channel-name>` or `--target channel:<channel-name>`)
- With a trigger connected to a broker (`--broker <broker-name>` or `--target broker:<broker-name>`)

As a *Developer* I want to update a connection

```
$ kn connection update myconnection  --filter <new filter>
```

As a *Developer* I want to see the details of a connection

Example

```
$ kn connection show myconnection

....
```

As a *Developer* I want to list all connections

```
# List all connections
$ kn connections list

NAME             SERVICE       TYPE       BROKER   FILTER  CHANNEL
myservice-001    myservice     importer
myservice-002    myservice     broker     default  ...     tempchannel
mysecondsrv-001  mysecondsrv                                mychannel
....

# List only connections which are attached to this service
$ kn connections list --service myservice
```

As a *Developer* I want to delete a connection

```
# Delete the connection
$ kn service delete-connection myconnection
```

Sequences

TODO: This section needs to be fleshed out with an opinionated, multi-step flow for building up a sequence.

As a *Developer* I want to build up a sequence interactively

- Interactive workflow by subsequent calls to an "append" or "insert" calls for adding transformer services.

As a *Developer* I want to update a sequence

As a *Developer* I want to list all sequences

As a *Developer* I want to remove a sequence

(with usage check)

As a *Developer* I want to see the details of a sequence

List of all transformers contained in the sequence

[1] This is currently still named as *describe* but under discussion to be renamed.

Custom types

There is a subtle difference between Knative serving and Knative eventing. Knative serving operates on a closed set of entities (`Service`, `Configuration`, `Revision` and `Route`) that is described by a fixed set of CRDs.

Knative eventing on the other hand is an *open* API which can be extended with custom types for channels and importers by registering CRDs and installing controllers which reconcile on those custom resources.

In the following we will stick to `Channel` resources. All what is said here can be mapped 1:1 to `Importers`, too, as both share the same characteristics with respect to custom types.

The kn client can easily query for all CRDs and match on all CRDs with a category of "knative" and "channel":

```
kind: CustomResourceDefinition
spec:
  group: messaging.knative.dev
  names:
    categories:
    - all
    - knative
    - messaging
    - channel
    kind: InMemoryChannel
```

The list of returned CRDs are the channels that can be used for creating new channels, whereby it is assumed that a corresponding controller has been installed on the server side, too.

However, since each channel type supports different configuration options, a client-side mechanism must allow a user to provide this configuration as command line options/flags.

For well-known types (like `InMemoryChannel`) the channel specific features are well known and can be directly supported by kn. For custom provided types, a plugin mechanism is required.

Such a channel plugin is an external binary placed in a well-known location (e.g. `~/.kn/plugins/channels/` or `~/.kn/plugins/importers/`)

The name of the binary reflects the type that should be used in `kn channel create --type <channel-type>` (e.g. `kn-channel-activemq` for a channel plugin managing channels of type "activemq")

The following commands given as arguments have to be supported by such a custom type plugin executable:

Table 1. Plugin contract for importer and channel plugins

Command	Description
`manifest`	Print out the CRD coordinates which connects this plugin to the CRD for which it is responsible. This can be a JSON structure with the kind, group and API version and a textual description of the channel type. Also, it should contain the list of possible options along with their descriptions so that a help message can be constructed and a validation can be happen on these options.
`create`	Create a resource of this kind. The provided command line arguments are handed through directly to the plugin. The first argument will be the name of the resource to create. The rest are options specific for this importer or channel.
`update`	Update a plugin-managed resource. The syntax is the same as for `create` except that a resource for the given name should be updated.
`describe`	Print out a human-readable description for an existing channel or importer.

If for one channel is either the CRD or the client side plugin is missing, then this channel type is disabled and does not show up on a `kn channel list`.

For the user, it should not matter whether the channel management is hardcoded in the kn binary or provided by a channel plugin. I.e. when listing all available channel types both types (internally provided, via plugin) are presented on the same level.

As already mentiones, the same mechanism should be implemented for importer plugins for handling custom importers, which are represented by CRDs in the same way as channels, with the difference that another naming convention is in effect (`kn-importer-twitter`) and those plugins might be stored in a different directory (`~/.kn/importers`).

The type discovery by querying matching CRDs requires that Knative eventing exposes the API operation for list CRDs also in its interface. If this is not possible, an alternative would be to make a pure client discovery by checking which plugins are installed. The combination of this list plus the list of well-known types is a list of supported types of this client. A client, however, would need then check whether the corresponding CRDs are registered on the server side, which can be done by a direct 'list' for such resources and checking for errors.

An alternative to plugins would be to evaluate the CRDs openAPI schema and provide a generic way to deal with these resources. This is a difficult task, but could be achieved, when some requirements on the CRDs registered could be imposed:

- There needs to be a mapping of possible CLI options (flat) to fields in the CR spec section (deep). This mapping could be e.g. attached as annotations to the CRD and provide a path like mapping to depict spec fields, which are mapped to annotation key which in turn are used as CLI options.
- Also mandatory option would need to be marked.
- It must be possible to create such resource in a generic fashion.
- For the detailed output either a generic layout is used for all such custom resources, or additional meta-data for printing out fields need to be added (similar to the mapping of options to CR fields).

The advantage of course is that there is no need for client side plugins.

For the sake of simplicity and flexibility, the plugin approach looks to be preferable (if the automatic distribution of channel and importer plugin can be easily achieved).

Examples

NOTE

To be done

CronJob Importer

Twitter Importer Plugin

References

- Kn Client issue tracking eventing integration
- Kn Eventing issue tracking UI/UX
- Scenarios for Knative Eventing

Glossar

Event topology

The concrete setup of Knative eventing with importers, channels and brokers managed by *Integrators* and used by *Developers*

Custom type

Type of channels and importers which are outside the set of well-known types

Channel plugin

A client-side plugin for a channel with a custom type

Importer plugin

A client-side plugin for an importer with a custom type